Efficient Checkpointing over Local Area Networks*

Avi Ziv
Information Systems Laboratory
Stanford University
Stanford, CA 94305-4055
E-mail: avi@isl.stanford.edu

Jehoshua Bruck
Mail Code 116-81
Pasadena, CA 91125
E-mail: bruck@systems.caltech.edu

Abstract

Parallel and distributed computing on clusters of workstations is becoming very popular as it provides a cost effective way for high performance computing. In these systems, the bandwidth of the communication subsystem (using Ethernet technology) is about an order of magnitude smaller compared to the bandwidth of the storage subsystem. Hence, storing a state in a checkpoint is much more efficient than comparing states over the network.

In this paper we present a novel checkpointing approach that enables efficient performance over local area networks. The main idea is that we use two types of checkpoints: compare-checkpoints (comparing the states of the redundant processes to detect faults) and store-checkpoints (where the state is only stored). The store-checkpoints reduce the rollback needed after a fault is detected, without performing many unnecessary comparisons.

As a particular example of this approach we analyzed the DMR checkpointing scheme with store-checkpoints. Our main result is that the overhead of the execution time can be significantly reduced when store-checkpoints are introduced. We have implemented a prototype of the new DMR scheme and run it on workstations connected by a LAN. The experimental results we obtained match the analytical results and show that in some cases the overhead of the DMR checkpointing schemes over LAN's can be improved by as much as 20%.

1: Introduction

Checkpointing schemes enable fault-tolerant parallel and distributed computing by leveraging the redundancy in hardware and software resources. The usage of checkpoints reduces the time spent in retrying a task in the presence of failures, and hence reduces the average completion time of a task [1]. Several authors, such as [3] and [4], proposed checkpointing schemes with task duplication for multiprocessor systems. These schemes use different techniques to achieve fault-tolerance and reduce the fault recovery time [6].

Traditionally, in parallel and distributed systems, each checkpoint in a checkpointing scheme serves two purposes. The first is to save the processor state and to reduce the fault-recovery time by supplying an intermediate correct state, thus avoiding rollback to the beginning of the task. The second purpose is fault-detection, which is achieved by executing the task on more than a single processor and comparing the processors states at each checkpoint.

The efficiency of checkpointing schemes is influenced by the time it takes to perform the comparisons and to store the states. Clearly, the performance can be improved if

*The research reported in this paper was supported in part by the NSF Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, by a grant from the IBM Almaden Research Center, San Jose, California and by a grant from the AT&T Foundation.

0-8186-6807-5/95 $4.00 © 1995 IEEE

the frequency of comparisons is low. On the other hand, frequent storing of states will facilitate efficient rollback after the detection of faults. The fact that checkpoints consist of both storing states and comparing between states – with conflicting objectives regarding the frequency of those operations – limits the performance of current checkpointing schemes.

This issue is especially important in systems where comparisons between the states of the processors require a much longer time than the storing of states. An example of systems with these characteristics are clusters of workstations connected by a LAN. In these systems, the bandwidth of the communication subsystem (using Ethernet technology) is roughly an order of magnitude smaller than the bandwidth of the local storage subsystem. Hence, storing a state in a checkpoint is much more efficient than comparing states over the network.

We present here a novel approach for checkpointing schemes that reduces the average execution time of a task by reducing the average rollback, without increasing the number of comparisons. In the proposed schemes there are two types of checkpoints, *compare-checkpoints (CCP)* and *store-checkpoints (SCP)*. The compare-checkpoints are used in the same way checkpoints are used in the traditional schemes, namely at each CCP the processors store and compare the states. In the second type of checkpoints, the store-checkpoints, the processors store their states without comparison. Taken together, the two types of checkpoints achieve both objectives of placing checkpoints, and hence improve the scheme performance.

We show how checkpointing schemes that use this approach can be analyzed, using as an example the DMR (Double Modular Redundancy) scheme. We use the analysis results to compare the average execution time of a task between the traditional DMR scheme and the proposed DMR scheme with store-checkpoints. The comparison results show that a significant reduction of up to 20% in the overhead time of the execution can be achieved when SCPs are used. The usage of the SCPs also allows an increase in the interval between compare-checkpoints, and hence reduces the needed synchronization between the processors.

We implemented the proposed DMR scheme with store-checkpoints on a UNIX based system, and measured the execution time of a test task using the scheme, when the task was executed on two SPARC IPX workstations connected by an Ethernet LAN. Comparison of the measured execution time with the calculated execution time shows a small difference, which is caused by the need to synchronize the workstations that executed the task at CCPs. Comparison of the measured execution time with different numbers of SCPs between CCPs reinforces the conclusion that using SCPs can result in a significant reduction in the overhead of the execution time.

In the next section we describe the concept of store-checkpoints and provide the analysis of the DMR scheme with store-checkpoints. In Section 3 the implementation of the DMR scheme with store-checkpoints is described, and the experimental results are shown.

2: Checkpointing schemes with store-checkpoints

In this section we show how store-checkpoints can be used to improve the performance of existing checkpointing schemes. To illustrate how the modification to the existing schemes is done, and to show how the modified schemes can be analyzed, we use the DMR (Double Modular Redundant) scheme. In this scheme the task is executed on two processors. At each checkpoint the states of the two processors are compared. If the states match, a correct execution is assumed, and the processors continue to the next interval. If the states do not match, both processors are rolled back to the previous checkpoint, and the execution of the same interval is repeated.

In the execution example in Figure 1, with the traditional DMR scheme, the states of the

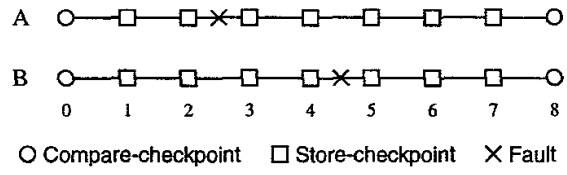


Figure 1. Execution of one interval with the DMR scheme and SCPs

processors at the end of the interval (checkpoint 8) do not match because both processors had faults. Hence both processors are rolled back to checkpoint 0 and the interval is executed again. In the proposed scheme seven SCPs are placed between the two CCPs. Those SCPs can be used to reduce the rollback to checkpoint 2 instead of checkpoint 0, and shorten the fault recovery period.

To make the search for the most recent matching checkpoint efficient, a binary search is performed on the Huffman tree [2] induced by the probabilities of rollback to each SCP. The average number of comparisons, $\overline{C}$, using the Huffman tree is approximately $\log_2 n$, when a CCP is placed every n checkpoints.

In the example in Figure 1 seven SCPs, numbered 1 to 7, are placed between CCPs 0 and 8. During the execution faults occur in both processors, and the comparison at checkpoint 8 fails. In the first step of the search for the most recent matching checkpoint, the states at checkpoint 4 are compared. The states do not match because processor A had an earlier fault. Next the states at checkpoint 2 are compared. Those states match. After that step we know that the required checkpoint is either 2 or 3. Finally the states at checkpoint 3 are compared. They do not match and a rollback to checkpoint 2 is done.

To analyze the average execution time of a task using the DMR scheme with store-checkpoints, we assume that a task of length 1 has to be executed. The task is divided into $m \cdot n$ intervals of length $t_I = \frac{1}{m \cdot n}$, and at the end of each interval a checkpoint is placed such that a CCP is placed every n checkpoints. The task is executed on two processors using the DMR scheme with store-checkpoints.

After a compare-checkpoint is reached and the fault recovery process is completed, the next CCP is placed n checkpoints from the last matching checkpoint. For example, in the execution in Figure 1, after the rollback to checkpoint 2, the next CCP is placed at checkpoint 10, and checkpoint 8 becomes a store-checkpoint.

The processors that execute the task are vulnerable to transient faults. The faults occur in each processor according to a Poisson random process with rate λ. The faults in the processors are independent of each other. We assume that faults can occur while the processors execute the task or store their state, but not while the states are compared.

The time to store the processors states is t_s, and the time to compare the states is t_{cp}. This time includes the time to rollback to the last matching checkpoint when a fault is detected.

Let c be the probability that no faults occurred in both processors, while executing a single interval or storing its states in the checkpoint that follows that interval. Because the faults in the processors are independent of each other, we can write the following expression for c:

$$c = \left(e^{-\lambda \cdot (\frac{1}{m \cdot n} + t_s)}\right)^2 = e^{-2\lambda \cdot (\frac{1}{m \cdot n} + t_s)}.$$

The probability that no fault occurred between the CCPs is simply c^n.

To calculate the average execution time of a task, we need to find the amount of time from the end of the previous CCP until the comparison and fault recovery of the current

CCP end. We also need to know how much progress in the execution of the task is achieved. We measure the progress as the number of intervals with matching states that are added.

To calculate the average progress achieved, we need to find where the last matching checkpoints are. Let X be the last matching checkpoints. X can get the values 0 (when the last matching checkpoints are the previous CCP), 1, ..., n. X gets the value $i, i < n$ if no faults occur in the first i intervals but a fault occurs in the $i+1$ interval, and the value n when no faults occur and the states of the processors at the current CCP match. So

$$\Pr(X = i) = \begin{cases} (1-c) \cdot c^i & i < n \\ c^n & i = n \end{cases}.$$

The average number of the most recent matching checkpoints, is

$$\overline{X} = \frac{c(1-c^n)}{1-c}.$$

The amount of time from the end of the previous CCP until the comparison and fault recovery of the current CCP end depends on whether faults occurred or not. If no fault occurred, the comparison at the CCP succeeds and and no fault recovery is needed. But when faults do occur, on average extra $\overline{C}$ comparisons are needed before the processors can be rolled back to the last matching checkpoint and execution resumed, where $\overline{C}$ is the average number of comparisons needed. So the average time from the end of the previous CCP until the comparison and fault recovery of the current CCP is

$$\overline{D} = \frac{1}{m} + nt_s + \left[1 + (1-c^n)\overline{C}\right] t_{cp}.$$

The average time to execute the whole task is

$$T = n \cdot m \cdot \frac{\overline{D}}{\overline{X}} = \frac{n(1-c)}{c(1-c^n)} \cdot (1 + mnt_s + m\left[1 + (1-c^n)\overline{C}\right] t_{cp}). \quad (1)$$

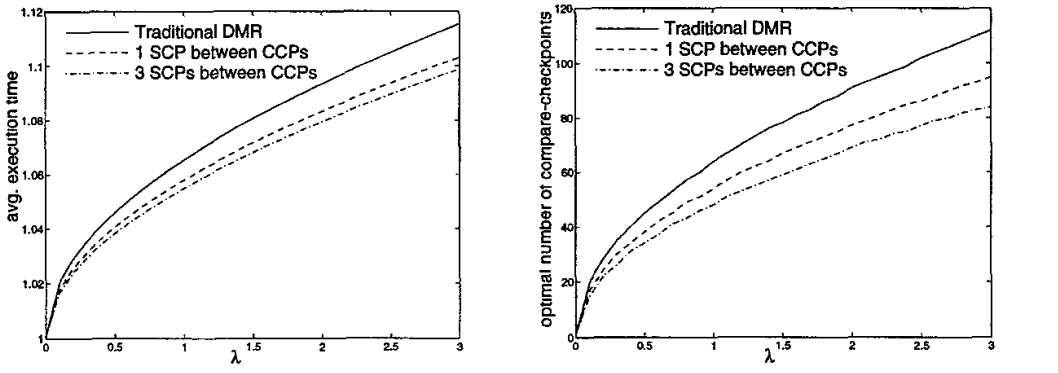
In Figure 2 the traditional DMR scheme performance is compared to the performance of the DMR scheme with CCPs every 2 or 4 checkpoints ($n = 2$ or $n = 4$). Figure 2a shows the average execution time of a task as a function of the fault rate λ. The times to store the processors states and compare them are $t_s = 10^{-5}$ and $t_{cp} = 5 \cdot 10^{-4}$. For each value of n and for each λ, the interval between CCPs is chosen such that the execution time is minimized. It can be seen from the figure that the usage of SCPs give a significant reduction in the overhead of the execution time.

In Figure 2b the number of CCPs that achieve the average execution time of Figure 2a is shown. The figure shows us that the usage of SCPs enables to place the CCPs farther apart and hence reduce the needed synchronization intervals between the processors.

3: Experimental results

To check the performance of the proposed DMR scheme with store-checkpoints, we implemented a prototype of the scheme on a UNIX based system, and measured the execution time of a task, using the proposed scheme, on two SPARC IPX workstations connected by an Ethernet LAN. In the implemented scheme each workstation saves its own state on its local disk. The comparison of the states is done by sending the state of one of the workstations to the other using a bidirectional reliable stream, based on the TCP protocol [5].

To reduce the amount of data sent on the network, only one workstation, called the slave, sends its state to the other workstation, called the master, for comparison. The



(a) average execution time (b) number of compare-checkpoints

Figure 2. Comparison between DMR scheme with and without store-checkpoints

master workstation, upon receiving the state from the slave, compares its own state with the received state, and sends further instructions to the slave based on the result of the comparison.

To compare the measured execution time of a task on the implemented scheme with the analytical results of Section 2, we first measured the time to perform the operations the scheme needs to achieve fault tolerance, namely storing the states of the workstations and comparing them. We found out that both the store and compare times grow linearly with the state size. The store rate we measured is about 9.9 Mbytes per second; while the compare rate is about 280 Kbytes per second.

The test task, which we used to check the performance of the scheme, consists of a block of memory that represents the state of the task. While the task executes, it continuously changes its state by performing read and write operations on it. The execution time of the test task, without checkpoints and faults, is 400 seconds.

We measured the execution time of the test task using the DMR scheme with store-checkpoints. In the test faults were generated synthetically according to a Poisson process, and each fault changes the value of a single random byte in the task state. The faults in the workstations are independent processes with identical rates.

Table 1 compares the measured average execution time of the test task to the analytical execution time derived using the results of Section 2. The comparison was done for two cases of state size; 100,000 bytes and 500,000 bytes. The times to store and compare the state, for the small state, are $t_s = 0.01$ second and $t_{cp} = 0.36$ second, respectively. For the large state these times are $t_s = 0.05$ second and $t_{cp} = 1.8$ seconds. In both cases the interval between compare-checkpoints is 8 seconds, and CCPs are placed every 4 checkpoints ($n = 4$). Table 1 shows, for each of the cases and for four different values of fault rate λ, the average measured execution time, the calculated execution time, and the difference between them.

Table 1 also shows that the analysis results are very close to the measured values, with an error of less than 3.5%. In all the cases the measured values are slightly larger than the calculated values. The reason for this difference is the need for synchronization of the two workstations at at each compare checkpoint.

Comparison of the measured average execution time of the test task with different numbers of store-checkpoints between the compare-checkpoints is shown in Table 2. The com-

Fault rate (faults/sec)	State size = 100,000 bytes			State size = 500,000 bytes		
	Measured time (sec)	Calculated time (sec)	Difference (%)	Measured time (sec)	Calculated time (sec)	Difference (%)
0.0025	441.49	432.09	2.2%	529.87	520.33	1.8%
0.0050	459.61	444.34	3.4%	554.80	540.92	2.6%
0.0075	465.39	456.77	1.9%	573.65	561.78	2.1%
0.0100	482.13	469.36	2.7%	601.15	582.90	3.1%

Table 1. Comparison between measured and calculated execution time

Fault rate (faults/sec)	Execution time				Overhead reduction
	$n = 1$	$n = 2$	$n = 4$	$n = 8$	
0.0025	446.50	441.48	441.49	**439.16**	15.8%
0.0050	467.48	460.37	**459.61**	460.88	11.7%
0.0075	485.08	478.94	**465.39**	470.29	23.1%
0.0100	503.45	487.07	**482.13**	483.63	20.7%

Table 2. Measured execution time for different cases of n

parison is done for task with state size of 100,000 bytes and with an interval of 8 seconds between the compare checkpoints. The table shows the execution time for 4 cases of n; 1, 2, 4, and 8. The bold entries in the table are for the lowest execution time for each fault rate. The last column in the table shows the percentage of reduction in the overhead of the execution time from the traditional scheme ($n = 1$) to the lowest average execution time.

The results in Table 2 reinforce the analytical results of Section 2 and show that using store-checkpoints between compare-checkpoints can reduce the execution time of a task. Using a small number of store-checkpoints between compare-checkpoints can reduce the execution time of a task by up to 5% and reduce the overhead of the execution time by 20%. The optimal number of store-checkpoints depends on the time to store and compare checkpoints, and the fault rate. The optimal number can be found using the analytical results given in Section 2.

References

[1] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21:546–556, June 1972.

[2] T. A. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley, 1991.

[3] J. Long, W. K. Fuchs, and J. A. Abraham. Forward recovery using checkpointing in parallel systems. In *The 19th International Conference on Parallel Processing*, pages 272–275, August 1990.

[4] D. K. Pradhan. Redundancy schemes for recovery. TR-89-cse-16, ECE Department, University of Massachusetts, Amherst, 1989.

[5] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1989.

[6] A. Ziv and J. Bruck. Analysis of checkpointing schemes for multiprocessor systems. In *The 13th symposium on Reliable Distributed Systems*, pages 52–61, October 1994.